DEV491
.NET Framework: Writing Faster Managed Code

Jan Gray
Architect
//blogs.msdn.com/jangr

Abhi Khune
Development Lead
//blogs.msdn.com/akhune

CLR Performance Team
Microsoft Corporation

Outline

- Performance engineering
- Managed code performance
  - Contrast with native code
  - Garbage collection
  - Common perf pitfalls
- Diagnosing perf problems w/ tools
- New perf features for VS 2005, Longhorn

Slow Software Stinks
Don't Ship It

- Symptoms
  - Locked-up UI, bad citizenship, poor scaling
- Causes
  - Mistakes in architecture, reuse, interfaces, data representations, algorithms
  - Not paying attention; low prioritization
    - Where are we? Where should we be?
- *Is* premature optimization the root of all evil?
  - *How to optimize the right stuff,*

Put Perf in Your Process
A Discipline that Works

- "That which gets measured gets done"
- *Set goals; Measure; Know your platform*
- Perf budgets, goals, *and automated tests*
- Process of "continuous" improvement
  - *Measure*, track, trend, refine
  - Cut failures
- Build a performance culture
  - User expectations

Moving to Managed Code

- Why? Productivity and quality
  - Do more with less code
  - Fewer bugs
  - Clean, modern libraries targeting modern requirements
  - Better software, sooner
- But is performance a problem?
  - Perception vs. reality

Managed Code
Close to the Machine

- Raw managed code *speed* rivals native code speed
- JIT compiler → optimized native code
  - Constant folding; constant and copy propagation; common subexpression elimination; code motion of loop invariants; dead store & dead code elimination; register allocation; loop unrolling; method inlining
- Very fast, self tuning garbage collector

demo

Comparing Managed and Native Code Performance:

1. JIT code quality
2. Object allocation perf

Managed Code
Close to the Machine, *continued*

- Low level cost model
  - → Jan Gray, *Writing Faster Managed Code: Know What Things Cost*, MSDN

- OK, so it's *fast* – but there are setbacks
  - Startup time, working set
    - Loading MSCORWKS, MSCORLIB, System.*.dll
    - JIT-compiling startup code path methods

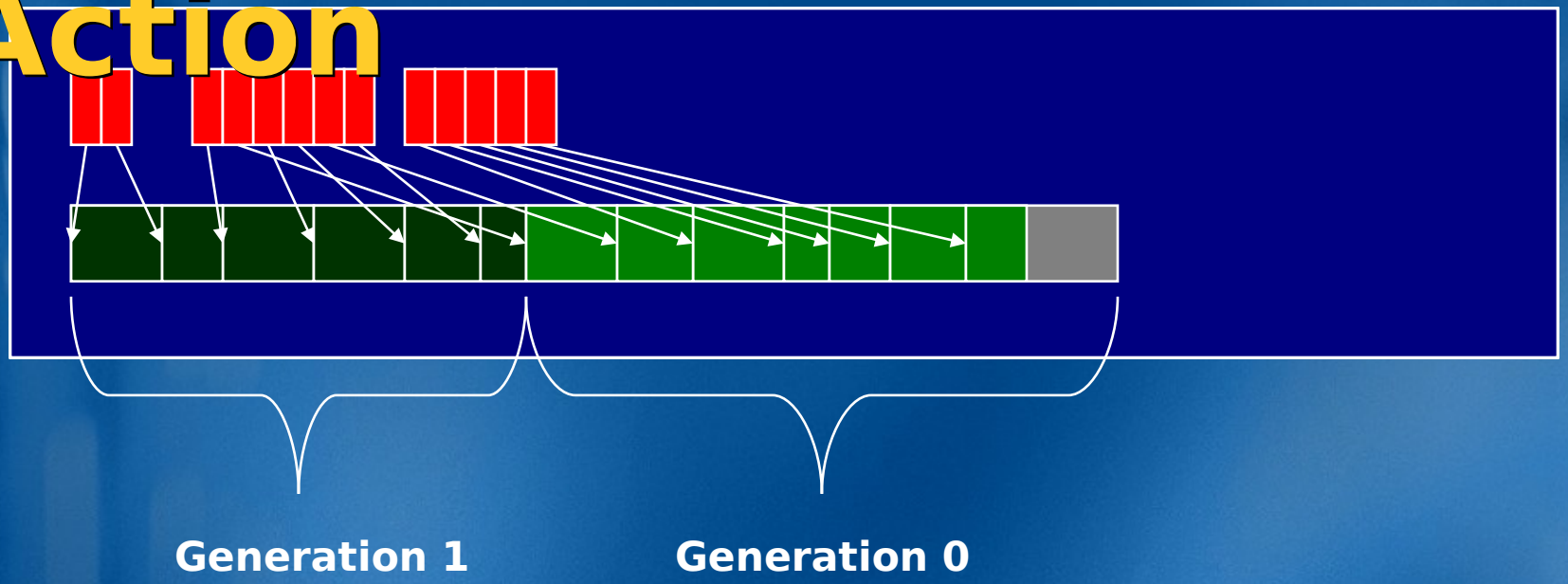
Yet Great Performance is Elusive

- The challenge of fast managed code
  - We're all newbies in managed code land
  - Learning *how* – not learning *how much*
  - Don't know what things cost
  - Everything is easier…
- The *trick* is to recognize, avoid pitfalls
  - New managed code pitfalls
    - E.g. automatic memory management

Know Your Garbage Collector

- *Why should I care?*
- GC concept
  - Automatic – no error prone manual freeing
  - Recycles useless unreachable objects
  - *new Type* triggers a GC; pause thread(s); trace & mark reachable objects from roots; compact live objects; recycle dead objects
- *Self tuning*
- Generational GC
  - Most objects are temporary, die young

Garbage Collection in Action



New objects allocated in generation 0

References to some temp objects are lost

GC: marks then compacts referenced (“live”) objects

GC: live object survivors promoted to next generation

Once again, new objects allocated in generation 0

Know Your Garbage Collector (2)

- Generational GC Heap
  - Gen0 – new objects
    - Cache conscious; very quick
  - Gen1 – objects promoted during gen0 GC
  - Gen2 – long lived objects, promoted during gen1 or gen2 GC
  - Large object heap
- Workstation GC
- Server GC
  - NUMA friendly; parallel; ASP.NET/hosted

Garbage Collection Pitfalls

- Object lifetimes still matter!
- Use an efficient “allocation profile”
  - Short lived objects are *cheap* (but not *free*)
  - And long lived objects don’t cause GC pressure
  - But don’t have a “midlife crisis” (avoid gen2 churn)
  - Review with perfmon counters, CLR Profiler
- Most common pitfalls
  - Inadvertently referencing “dead” object graphs
    - Null out object references (where appropriate)
  - Implicit boxing
  - Pinning lots of young objects

Garbage Collection Pitfalls (2)

Finalization and the Dispose Pattern

- Finalization: ~C()
  - Native resource clean up – nondeterministic
  - GC; object unref'd; promote; queue finalizer
  - Costs: retains object and its object graph; finalizer thread; bookkeeping; call
  - Expensive – use sparingly!
  - Try to refer only to the native resource
- Dispose pattern: implement and use

Pitfall: Indiscriminate Code Reuse

- Your choices determine your perf
  - Your architecture, algorithms, …
  - Your uses of .NET FX types and methods
- We keep seeing poor choices
  - No API is cheap enough to use everywhere
- Does *ease of reuse* cloud our judgment?
  - The useful *friction* of native code reuse
- You have to do your homework

Data Pitfalls

- Data locality
  - Remote → disk → DRAM → cache
    10^7-10^9 → 10^7 → 100 → 1 ns
  - GC: objects allocated together in time will stay together in space
- Data representation – XML
  - *Finally, a universal solvent…*
  - *"It's not just for interop anymore"*
  - Uncompetitive performance compared to binary record formats
    - System.Configuration may read *and parse* 100s of KB of XML to find a few flags settings
  - Fashionable – or fast?

Reflection Pitfalls

- *Powerful dynamic metaprogramming facilities*
- GetType is fine, but *is*, *as* even better
- GetMember/Method/Field, Get/SetValue, Invoke, CreateInstance
  - ~100X slower than the direct way
  - Footprint of metadata and reflection objects
- Insidious
  - .NET FX code that uses reflection
  - Late bound code in VB.NET, JScript.NET
  - Enforce early binding
    - `Option Explicit On`
      `Option Strict On`
- Rewrite to avoid reflection in critical

P/Invoke, COM Interop Pitfalls

- *Embracing legacy native code reuse*
- Efficient, but frequent calls add up
- Costs also depend on marshaling
  - Primitive types and arrays of same: cheap
  - Others, *not*; e.g. Unicode to ANSI strings
- Diagnosis
  - Perfmon: .NET CLR Interop counters
  - Time based profiling

Deployment Pitfalls

- Assemblies
  - Units of deployment, encapsulation, CAS, versioning, servicing, loading, separate authoring – so many dimensions!
  - Performance-wise: the fewer, the better!
- Know which assems are loaded, when, and why
- *GAC install* your signed assemblies
  - Avoids repetitive SN signature verification
- NGEN – ahead-of-time native code generator
  - Caches native code image DLL
  - *May* reduce startup time, improve page shareability
  - (Currently) code may run slower
  - Try it and measure for yourself

Analyzing Performance Problems
Code Inspection

- Ildasm – findstr “box”
- Debuggers – Module loads, rebasing
- *FxCop* – Static Analyzer

Analyzing Performance Problems
Measure It, With Tools

- High level diagnostics
  - Taskmgr, perfmon, vadump, event tracing for Windows (ETW)
- Time
  - Code profilers, timing loops, ETW
- Space
  - CLR Profiler, UMDH, code profilers, ETW

demo

Performance Tools: Diagnosing and Fixing Performance Problems

Analyzing Performance Problems Demo Recap

- Top down analysis
- Perfmon
- CLR Profiler
  - Peter Sollich, *CLRProfiler*, MSDN

“What’s Coming?”

- Some VS 2005 perf features
- Looking ahead to Longhorn and ubiquitous managed code

Some VS 2005 CLR Perf Work

- Cheaper AppDomains, faster cross-AD calls
  - ~1.1X-50X faster
- Faster delegate create (~10X) and invoke (~2X)
- Generics and new generic collections
  - No boxing, no casting, valuetype enumerators
- Lightweight Reflection.Emit
  - Per method granularity
  - GCs method bodies
- GC improvements
  - Memory pressure API

Looking ahead to Longhorn

Nearly Ubiquitous Managed Code

- CLR and Longhorn OS assemblies will be loaded into many processes
  - Must minimize the marginal time and space costs of each new managed process
  - NGEN-centric strategy
  - Reduce managed modules' private pages *and* working set by 50% each (!)
  - Reduce *CLR* startup time by >50%

Conclusions / Call To Action

- **Aim high!**
  - ***Set goals; measure; and know your platform***
- **Watch out for pitfalls**
  - **Memory mgmt; code reuse; reflection; interop; XML; exceptions; deployment …**
- **Measure!**
  - **Obtain and master the tools you need**
- **Towards a great platform for both RAD *and hard core systems***

Resources

- ***Patterns & Practices: Improving .NET Application Performance and Scalability*** [http://msdn.microsoft.com/**perf**]
- **CLR Performance website** [http://www.gotdotnet.com/team/clr/about_clr_performance.aspx/]
  - **Links to MSDN .NET Dev Center Perf Papers**
  - ***Know What Things Cost***
  - ***Thinking about Performance,* HeadTrax report**
  - **CLR Profiler**
- **news:microsoft.public.dotnet.framework.performance**
- **Blogs** [http://blogs.msdn.com/]

Session Evaluation

Please fill out a session evaluation on CommNet

Q1: **Overall satisfaction with the session**

Q2: **Usefulness of the information**

Q3: **Presenter's knowledge of the subject**

Q4: **Presenter's presentation skills**

Q5: **Effectiveness of the presentation**

Microsoft®

Your potential. Our passion.™

Microsoft Tech·Ed 2004